Ch6: Arduino Programming – Part 2

Contents:

Analog I/O

Serial Interfaces

Analog I/O

Arduino Programming: Analog I/O

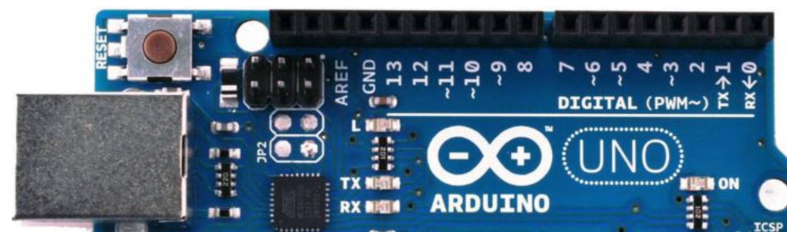
Till now, you are able to generate **digital outputs** from your Arduino. But what if you want to output a voltage other than 0V or 5V?

- Using the Digital-to-Analog Converter (DAC) pins on Arduino Due not Uno.
- Using an external Digital-to-Analog Converter (DAC) chip.
- Using a *trick* called **Pulse-Width Modulation** (PWM) to emulate an analog signal.

PWM is a technique for getting analog results with digital means.

Applications: Fading LEDs, Changing speed of motors,... .

Pins which are able to emulate analog signals are marked with a ~ on the board, i.e., 3, 5, 6, 9, 10, and 11.



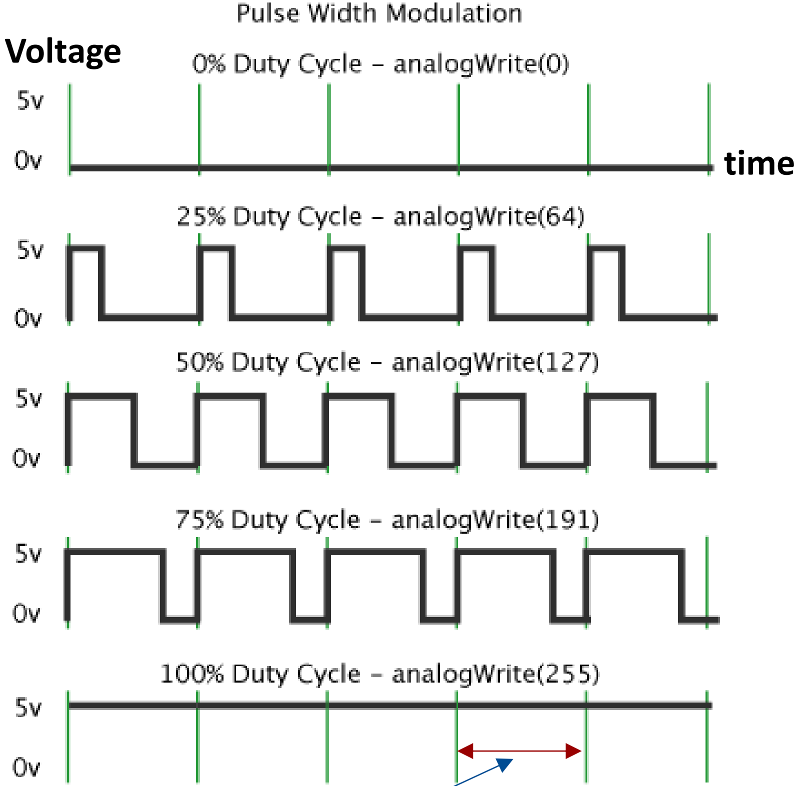
Pulse-Width Modulation (PWM)

Digital control is used to create a **square wave**, a signal switched between On and Off.

This On-Off pattern can simulate **Analog-like** (Analogish) voltages in between full On (5 V) and Off (0 V) by changing the portion of the time the signal spends On versus the time that the signal spends Off.

Pulse Width: The duration of "On time".

Duty Cycle: the percentage of time that a square wave is high versus low.



Note: The frequency of the PWM signal on most pins is approximately 490 Hz, but on the pins 5 and 6 of Uno is approximately 980 Hz (i.e., PWM **period** is about 2 ms or 1 ms).

Fading an LED

The PWM output of Arduino Uno is an 8-bit value, and you can write values from 0 to $2^8 - 1$ (255) to simulate voltages from 0V to 5V.

PWM pins use analogWrite() command to emulate analog signals:

analogWrite(pin , value);

Pin Number (pin)

A value from 0 to 255 (0 to 2^8-1) to represent Duty Cycle (value)

```
const int LED=9;
void setup() {
  pinMode (LED, OUTPUT);
}
void loop() {
  for (int i=0; i<256; i++) {
    analogWrite(LED, i);
    delay(10);
  }
  for (int i=255; i>=0; i--) {
    analogWrite(LED, i);
    delay(10);
  }
}
```

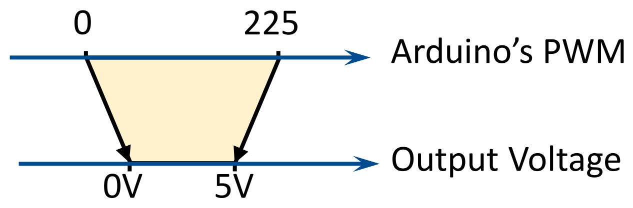
To fade the LED up (first for loop)

To fade the LED down (second for loop)

`i--` equivalent to `i=i-1`

`i++` equivalent to `i=i+1`

Digital: Value is either HIGH or LOW
Analog: Value ranges from 0-255 (for output) and 0-1023 (for input)

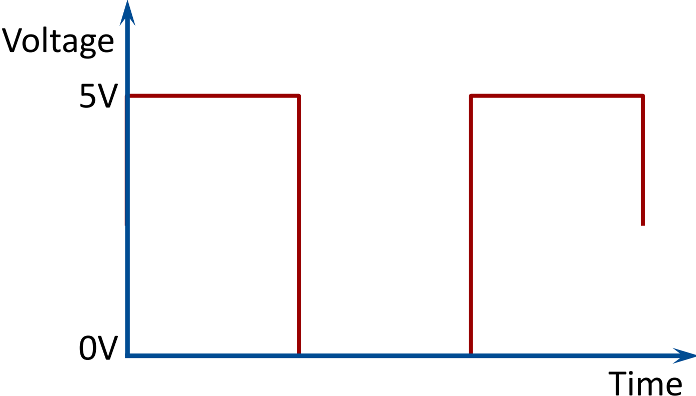


The LED is switching On and Off fast enough. Since it is blinking faster than your eyes can perceive, the result is **as if** the signal is a steady voltage between 0 and 5v controlling the brightness of the LED.

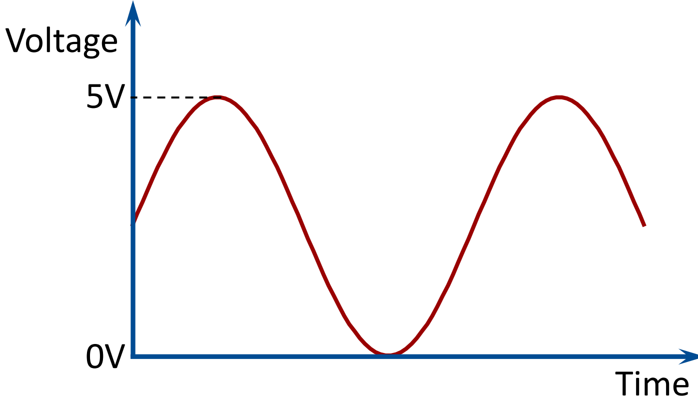
Arduino Programming: Analog I/O

The world around us is **analog**. *Ex.*: Temperature, Sunlight Brightness, Wind Speed, …

Digital Signal (e.g., Square Wave)
It varies between only two values: 0 and 5V like a Push Button.



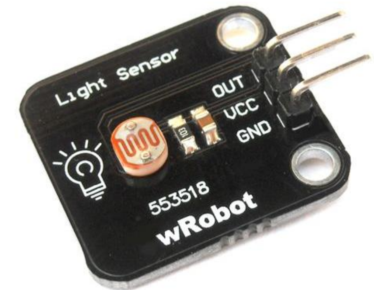
Analog Signals (e.g., Cosine Wave)
It includes an infinite number of values between those two voltages: 0 and 5V.



Since all devices only process digital signals, in order to interface with the real world, analog signals need to be converted to digital signals using **Analog-to-Digital Converters** (**ADCs**).

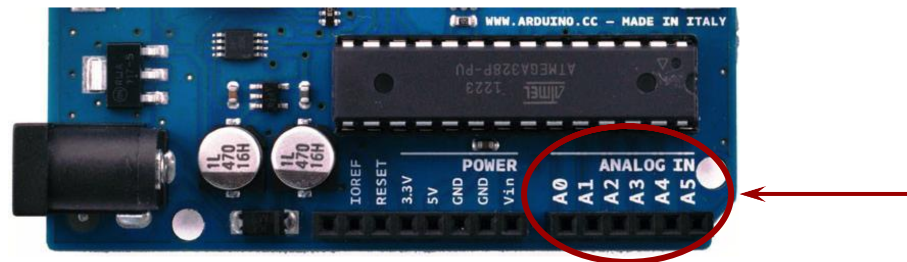
Arduino’s ADCs

Suppose that you want to measure the **brightness** of a room using a light sensor which can produce a **varying output voltage** that changes with the brightness of the room.



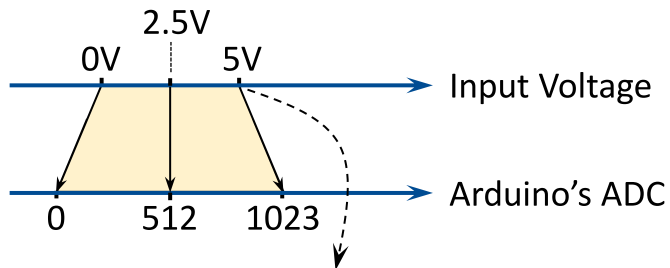
How do you **read** those values with an Arduino to figure out how bright the room is?

The Arduino’s Analog-to-Digital Converter (ADC) pins are used to convert analog voltage values into number representations that you can work with.



Arduino’s ADCs

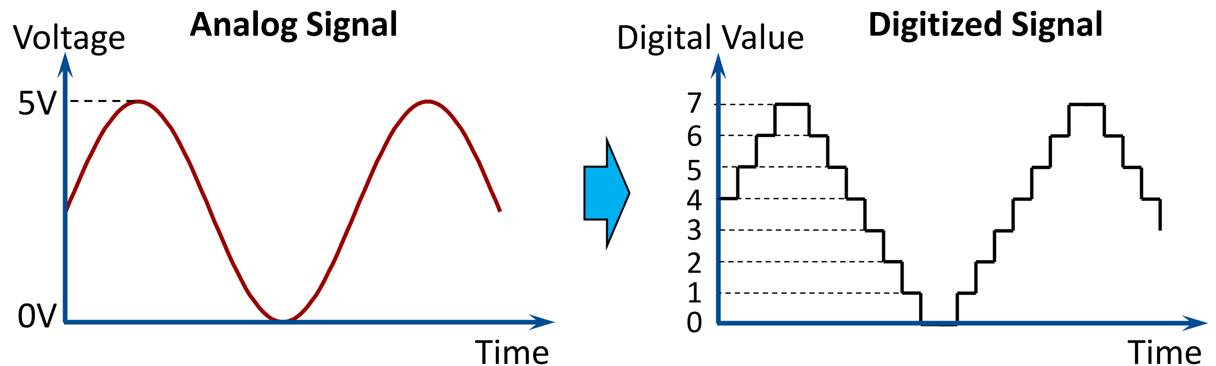
The **accuracy** of an ADC is determined by the **resolution**. The ADC of Arduino Uno is 10 bits and can subdivide (or quantize) an analog signal into 2^{10} (=1024) different values.



The Arduino can **linearly** assign a value from 0 to 1023 for any analog voltage value that you give it from 0V to 5V.

Reference Voltage for Arduino ADC pins (the max voltage that you are expecting on these pins) is 5V by default (and it is possible to be changed by connecting AREF to a lower voltage).

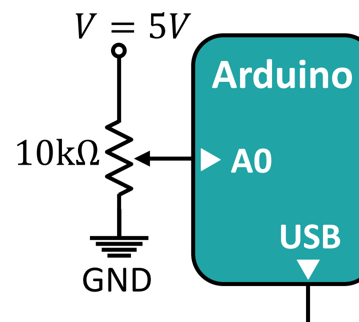
To better understand, consider a 3-bit ADC which can quantize an analog signal into 2^3 (=8) different values, from 0 to 7.



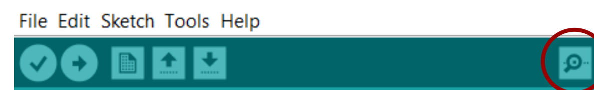
Reading a Potentiometer

Arduino reads potentiometer's values and transmits them via the USB connection to the serial terminal on your computer.

```
const int POT = A0;
int val = 0;
void setup() {
  Serial.begin(9600);
}
void loop() {
  val = analogRead(POT);
  Serial.print("Value: ");
  Serial.println(val);
  delay(500);
}
```



As you turn the potentiometer, you vary the voltage that you are feeding into analog input A0 between 0V and 5V.



Code Description

```
const int POT = A0;
int val = 0;
void setup() {
  Serial.begin(9600);
}
void loop() {
  val = analogRead(POT);
  Serial.print("Value: ");
  Serial.println(val);
  delay(500);
}
```

A variable to **hold** the analog value reading from the potentiometer.

Arduino's serial communication must be started in the **setup()** as

Serial.begin(BaudRate)

Baud Rate which specifies the communication speed (the number of bits being transferred per second).

Note: 9600 baud is a common value.

ADC Pin Number

Value of Arduino's analog pin is read using analogRead(pin)

Code Description

A value or text can be printed over serial to the computer's serial terminal using

Serial.print(arg)

A Value or "Text"

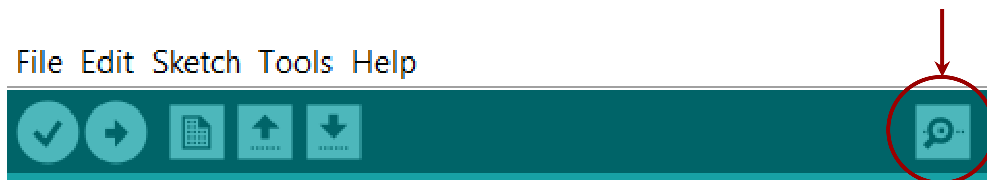
The command Serial.println() is similar to Serial.print() but followed by a ***newline*** that advances the cursor to the next line.

```
const int POT = A0;
int val = 0;
void setup() {
  Serial.begin(9600);
}
void loop() {
  val = analogRead(POT);
  Serial.print("Value: ");
  Serial.println(val);
  delay(500);
}
```

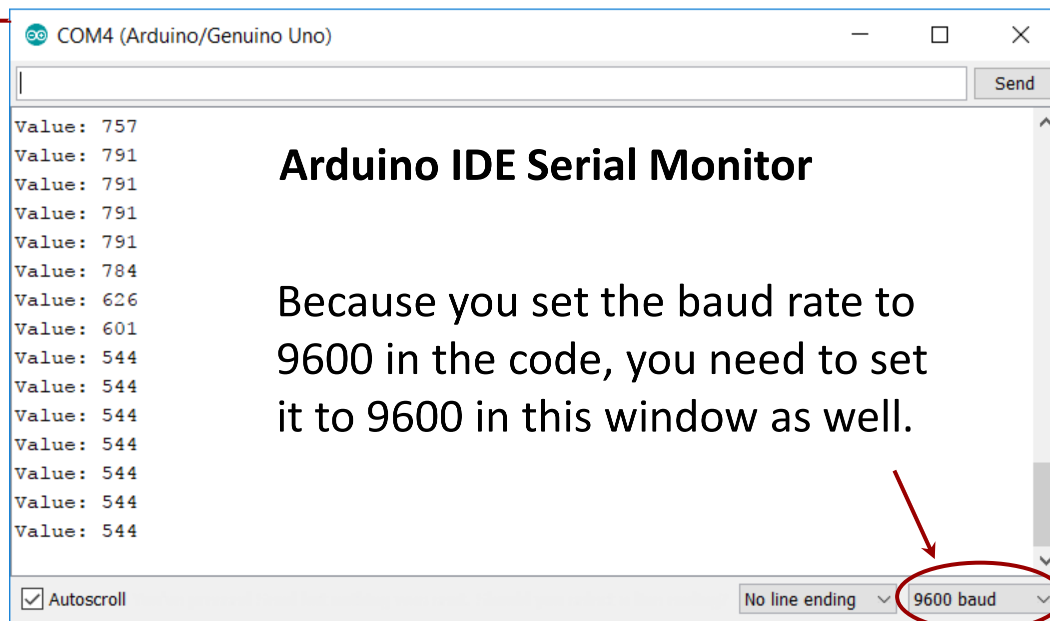
The delay() is used to make reading the data which is transmitted to the serial terminal easier. Hence, Arduino transmits data via the USB connection to the serial terminal on your computer every 500ms and TX LED on Arduino blinks every 500ms.

Arduino IDE Serial Monitor

In order to launch Arduino IDE Serial Monitor and see what Arduino is sending, click the circled button as shown.



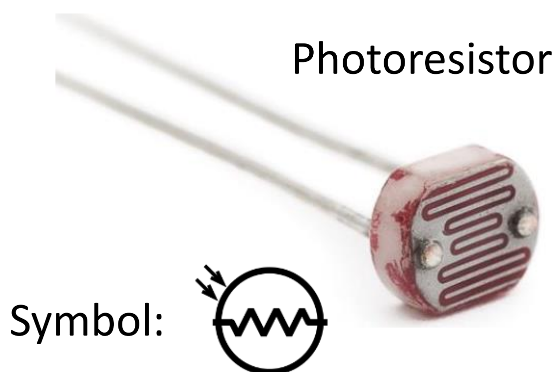
By turning the potentiometer dial, you will see the numbers go up and down between 0 and 1023 to correspond with the position of the potentiometer.



Resistor-Based Sensors

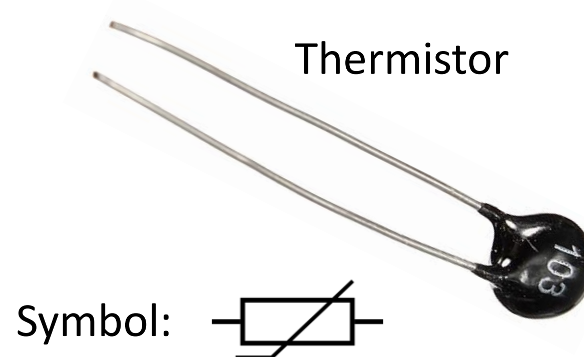
Some materials **change resistance** as a result of physical action. For example, some semiconductors change resistance when struck by light (**Photoresistors**), and some polymers change resistance when heated or cooled (**Thermistors**).

Photoresistor



Symbol:

Thermistor



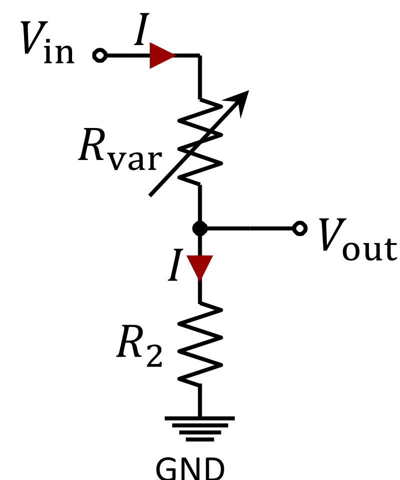
Symbol:

Because these sensors are changing resistance and not voltage, we need to create a **voltage divider circuit** so that we can measure their resistance change.

Adjustable Voltage Divider

An adjustable voltage divider can be made from **a fixed resistor** and **a variable resistor**.

In voltage dividers, if one of the resistors is a variable resistor (like resistor-based sensors including photoresistors and thermistors), the change in output voltage (V_{out}) that results from the varying resistance (R_{var}) can be monitored.



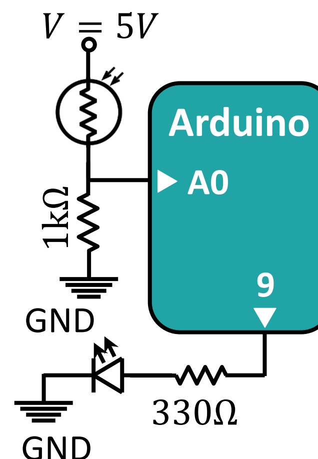
$$I = \frac{V_{\text{in}}}{R_2 + R_{\text{var}}} \longrightarrow V_{\text{out}} = R_2 I = \frac{R_2}{R_{\text{var}} + R_2} V_{\text{in}}$$

The size of the other resistor (R_2) can be used to set the **sensitivity** of the circuit, or a potentiometer can be used to make the sensitivity **adjustable**.

Light Indicator

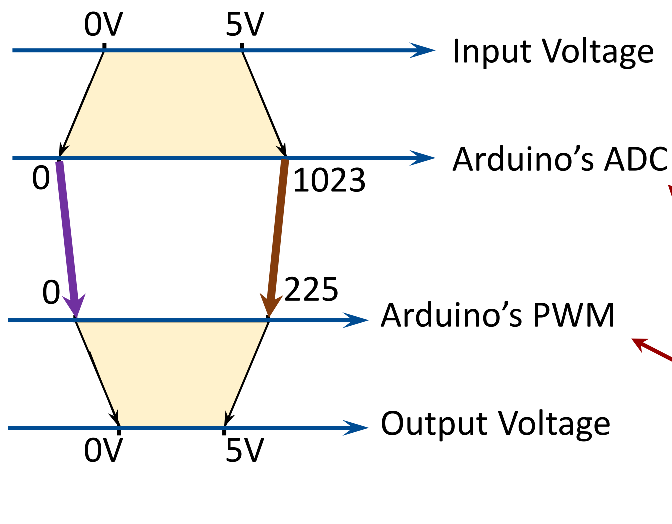
The LED gets brighter as the room gets brighter.

```
const int LED=9;
const int LIGHT = A0;
int val = 0;
void setup() {
  pinMode(LED, OUTPUT);
}
void loop() {
  val=analogRead(LIGHT);
  val=map(val, 0, 1023, 0, 255);
  analogWrite(LED, val);
}
```



When in complete darkness, its resistance is maximum and when saturated with light, its resistance drops nearly to zero.

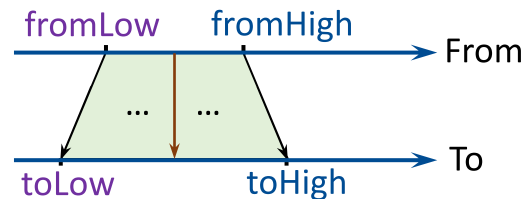
Code Description




```
const int LED=9;
const int LIGHT = A0;
int val = 0;
void setup() {
  pinMode(LED, OUTPUT);
}
void loop() {
  val=analogRead(LIGHT);
  val=map(val, 0, 1023, 0, 255);
  analogWrite(LED, val);
}
```

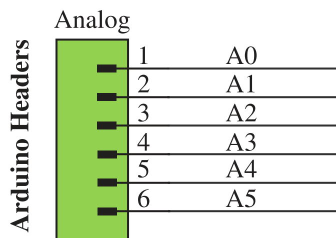
map(value, fromLow , fromHigh , toLow , toHigh)

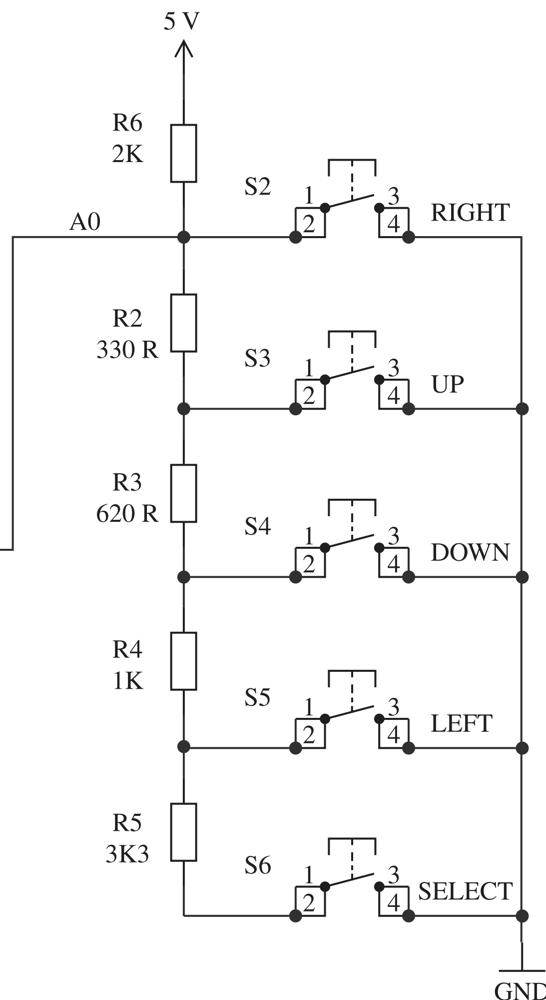
map() command linearly **maps** a number from one range to another. That is, a **value** of **fromLow** would get mapped to **toLow**, a **value** of **fromHigh** to **toHigh**, **values** in-between to values in-between, etc.



Multiple Switches to One Analog Input

If you have a lot of switches and do not want to tie up a load of digital inputs, then a common technique is to use an analog input and a number of resistors. The voltage at the analog input will then depend on the switches that are pressed.

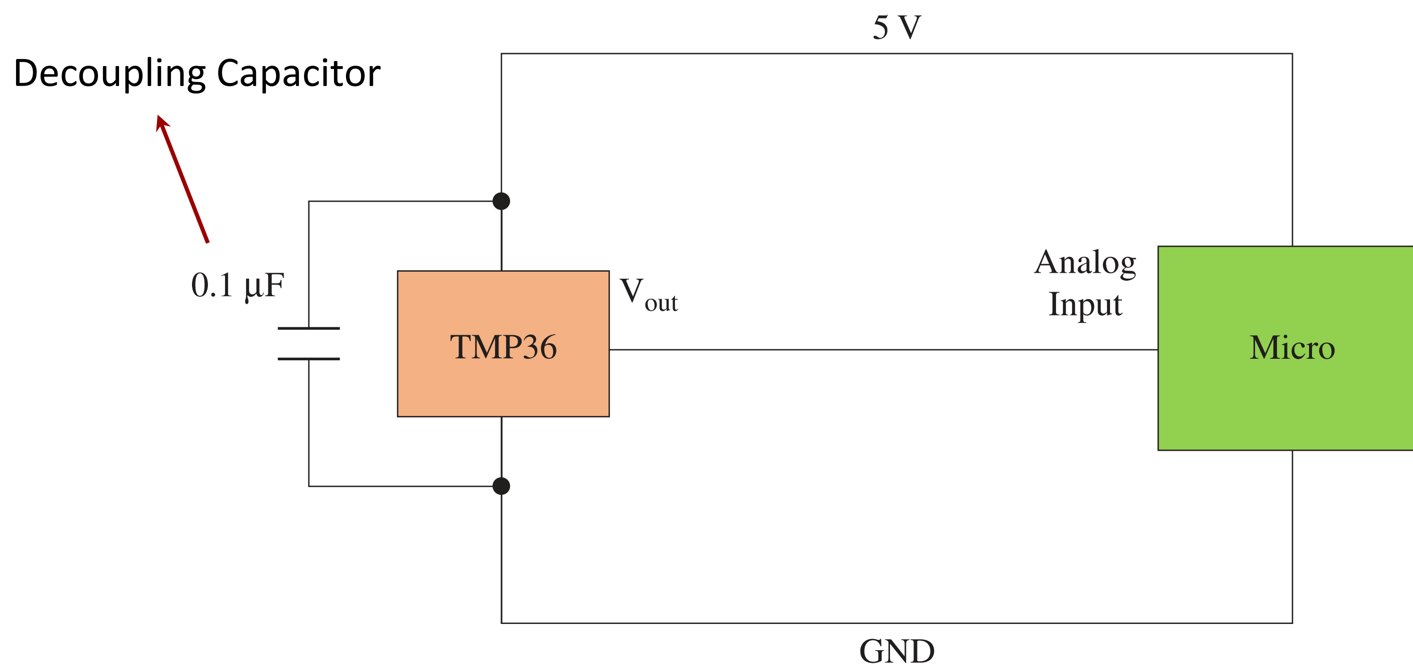




Note that in the code, you need to specify a range to indicate a certain button, rather than just one value.

Reading Analog Output of Sensors

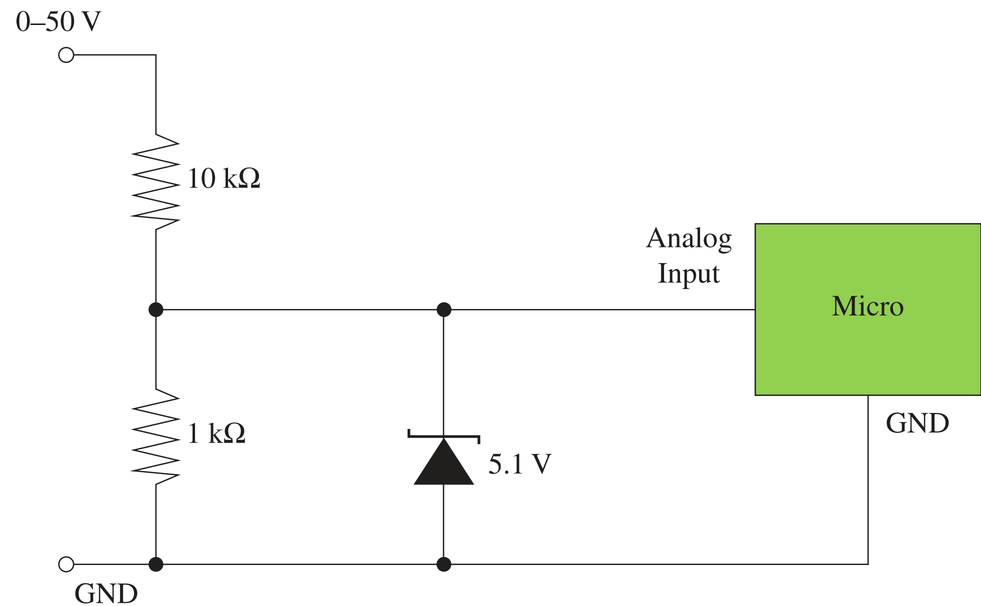
Many of the sensors provide an analog output to indicate the property that they are reading, e.g., the TMP36 temperature sensor IC.



Reading Analog Output of Sensors

If you are measuring a voltage (say 0-50 V) that is outside the range of the microcontroller's analog input (say 0-5 V), then you can just use two resistors as a voltage divider to reduce the voltage appropriately.

- If there is a risk that the voltage may exceed the expected range (0-50 V), you can protect the microcontroller's analog input by adding a Zener diode.

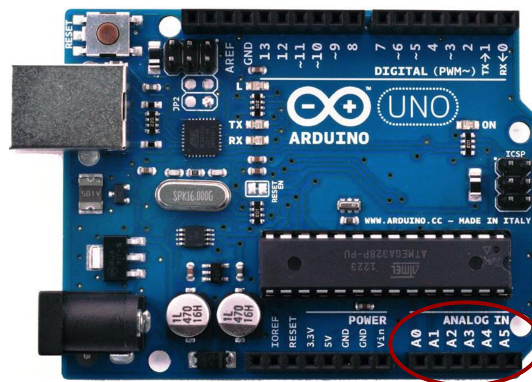


Using Analog Pins as Digital Pins

The **analog pins** can be used **identically** to the **digital pins**, using the aliases A0 (for analog input 0), A1, A2, A3, A4, and A5.

For example, the code would look like this to set analog pin A0 to an output, and to set it HIGH:

```
pinMode(A0, OUTPUT);
digitalWrite(A0, HIGH);
```



Serial Interfaces

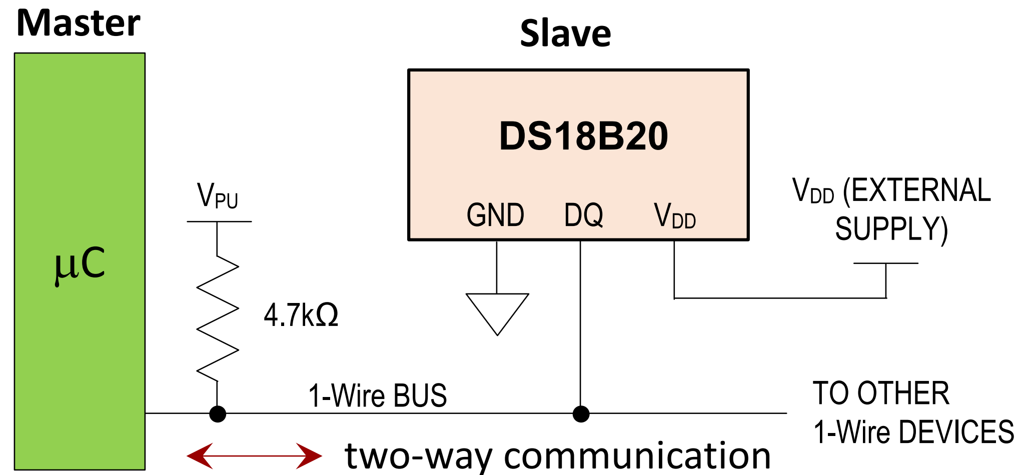
Serial Interfaces

There are a number of different standards for serial interfaces to microcontrollers, which use different numbers of pins and approaches to communication.

- **1-Wire Bus**
- **I^2C Bus, also known as the Two-Wire Interface (TWI)**
- **Serial Peripheral Interface (SPI) Bus**
- **USART Serial**

1-Wire Bus

The **1-Wire Serial Bus** uses just **a single connection** to communicate. Up to 255 devices can be connected to the same wire. This standard was developed by Dallas Semiconductors and is used in a variety of devices, e.g., DS18B20 temperature sensor.

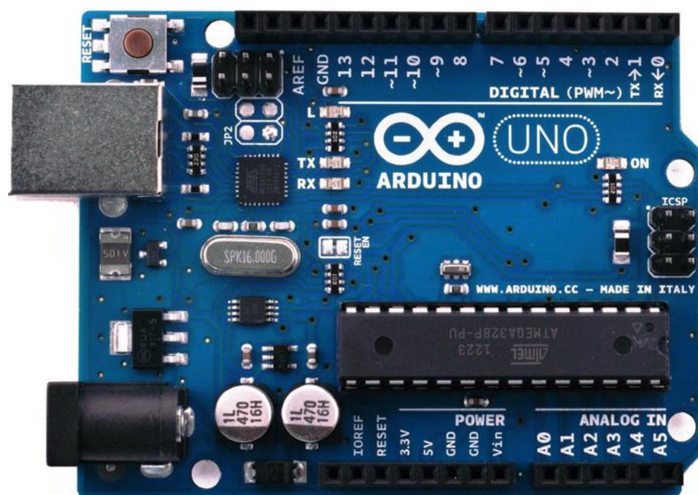


- 1-Wire devices act as either a **master** or **slave**. The microcontroller will be the master, and the peripheral devices, such as sensors, the slave.
- Every slave device has a unique 64-bit identifier that is programmed into ROM by manufacturer.

1-Wire Bus

The MCU uses the pin as both an input and an output (the pin's direction changes while the program is running). Communication is always initiated by the master and send a command as a sequence of pulses to find all the devices IDs connected to the data line (by a special search protocol) and communicate with them.

Any available digital pin can be used for 1-Wire communication.



In Arduino IDE, **OneWire.h** library is available for 1-wire communication. It hides the low-level timing and make the programming easier.

I²C Bus

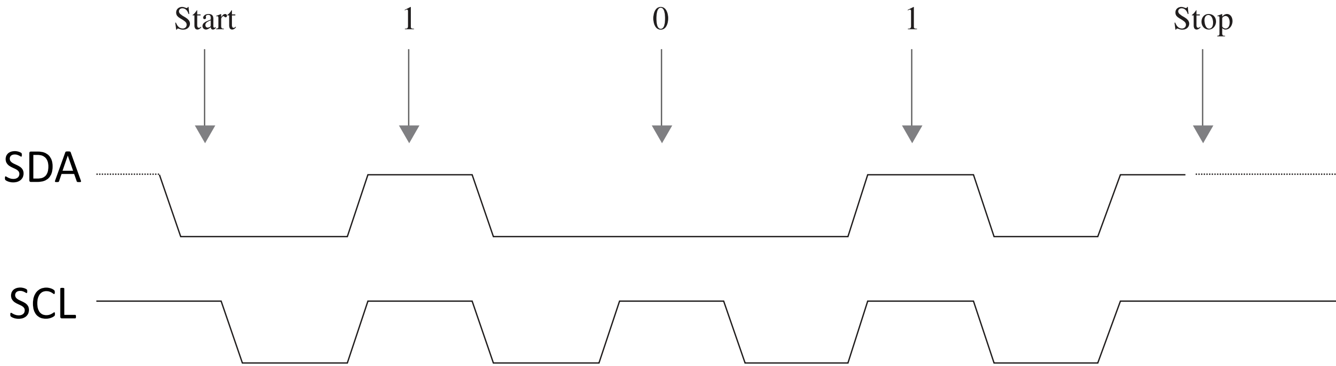
I²C, also known as the **Two-Wire Interface** (**TWI**), serves much the same purpose as 1-Wire, although it has two wires rather than one for data.

- It can support multiple devices connected to the same two wires.
- The **Serial Clock Line** (**SCL**) is a clock and the **Serial Data Line** (**SDA**) a bidirectional data line.
- I²C devices are either masters or slaves. There can be more than one master device.
- Microcontrollers can use I²C interface to exchange data with other microcontrollers.
- Since all the devices are receiving all the messages, each I²C slave device must have a **unique 7-bit address**, or **ID number**.

- I²C is faster than 1-Wire, with top speeds of up to 400 kbits/s.

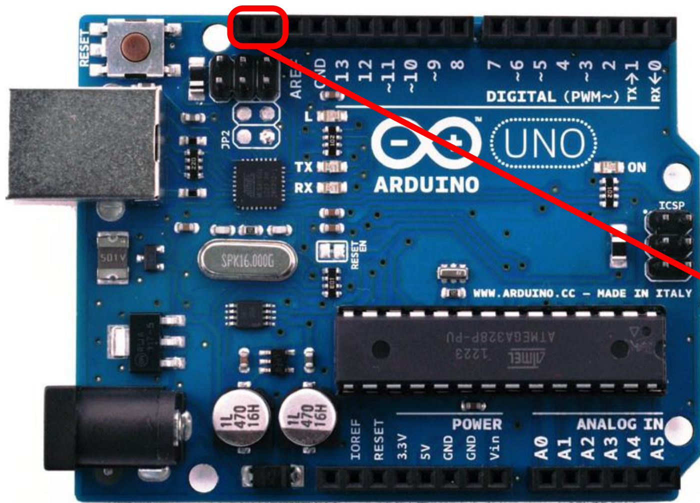
I^2C Bus

The master supplies the SCL clock, and when there is data to be transmitted, the sender (master or slave) takes the SDA line out of tri-state and sends data as logic highs or lows in time with the clock signal. When transmission is complete, the clock can stop, and the SDA pin be taken back to tri-state.



How I2C Communication Works

I^2C Bus



I^2C (TWI) Interface

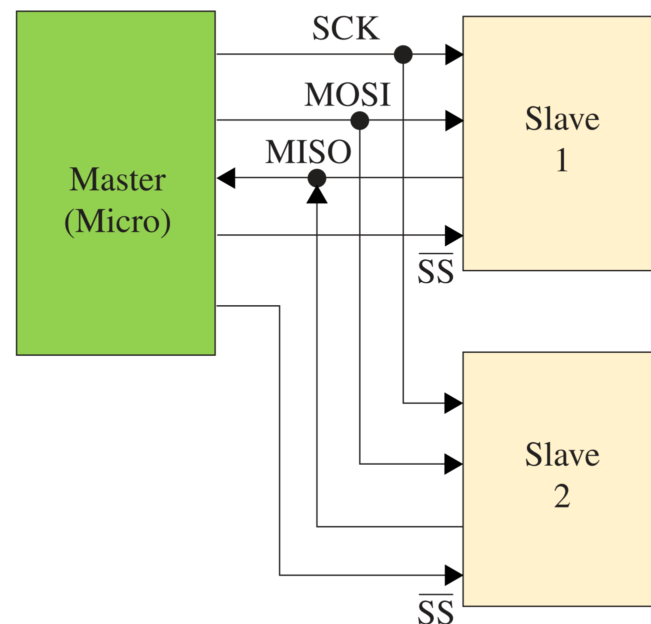
It actually uses two of the analog pins (A4 or SDA pin and A5 or SCL pin).

In Arduino IDE, **Wire.h** library is available for I^2C (TWI) communication. It hides the low-level timing of the protocol and make the programming easier.

Serial Peripheral Interface (SPI)

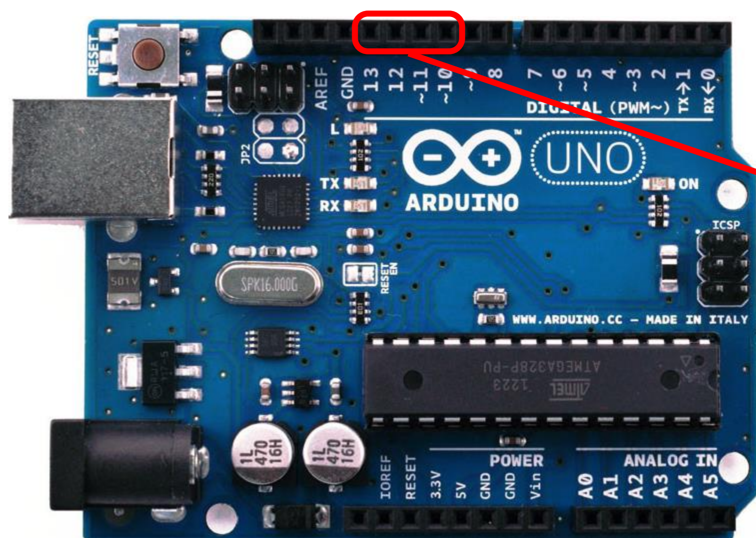
Serial Peripheral Interface (**SPI**) uses four data lines and is faster than 1-Wire and I^2C buses (up to 80 Mbits/s).

- There can only be one master device.
- The slave devices are not assigned addresses. Instead, the master must have a dedicated **Slave Select** (**SS**) line for each of the slave devices, just selecting the one it communicates with.
- The other extra line is required because separate lines are used for each direction of communication.



- The **Master Out/Slave In** (**MOSI**) line carries the data from the master to the slave device, and the **Master In/Slave Out** (**MISO**) line does the reverse. The **Serial Clock** (**SCK**) is the clock pulses which synchronize data transmission generated by the master.

Serial Peripheral Interface (SPI)



ISP Interface:
10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK)

In Arduino IDE, **SPI.h** library is available for ISP communication. It hides the low-level timing of the protocol and make the programming easier.

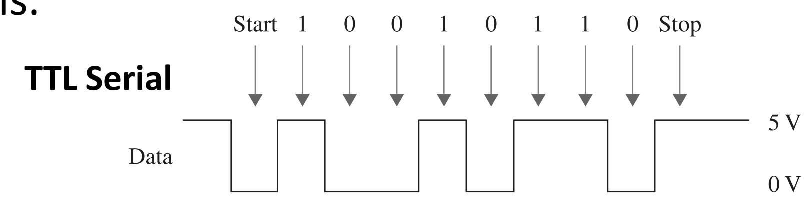
- Note that SPI is also used as a means of ICSP (In Circuit Serial Programming) on some microcontrollers, e.g., ATmega and ATtiny families.

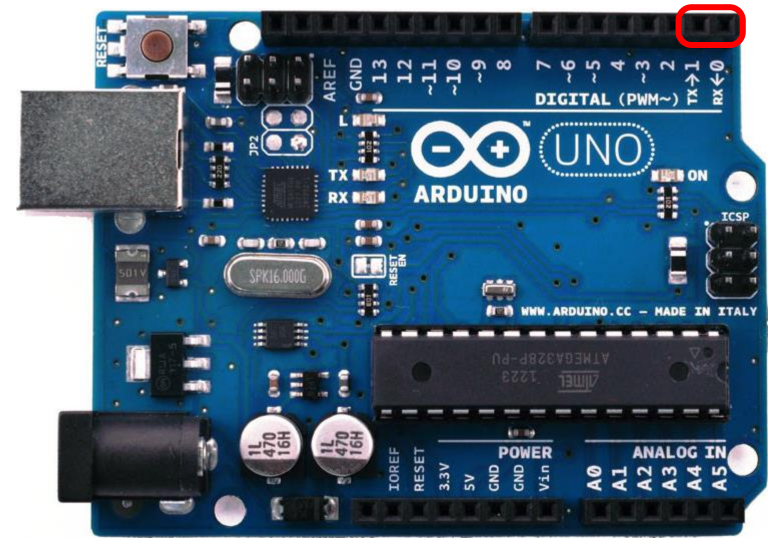
USART Serial

Using microcontroller's integrated UART/USART, you can send information between the microcontroller and your host computer, or other serial-enabled devices (including other microcontrollers).

ATMega328P microcontroller (on the Arduino Uno) has only one hardware serial port (USART). It includes Transmit (TX) and Receive (RX) pins that can be accessed on digital pins 0 and 1 of the Arduino Uno. Serial communication on pins TX/RX uses TTL logic levels.

TTL Serial





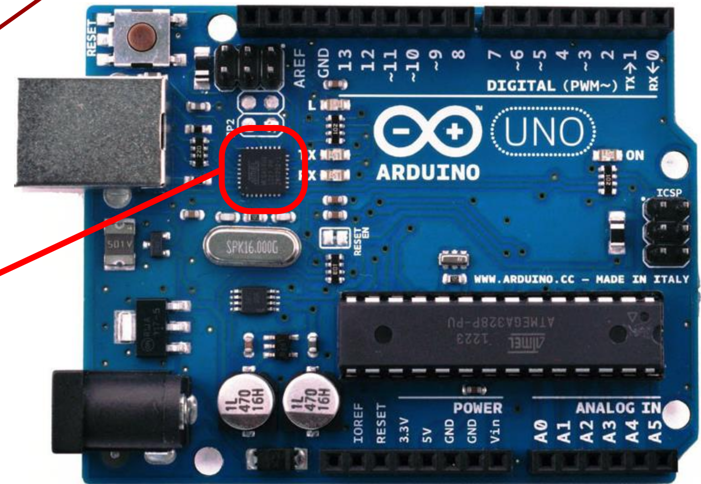
This serial interface is used for

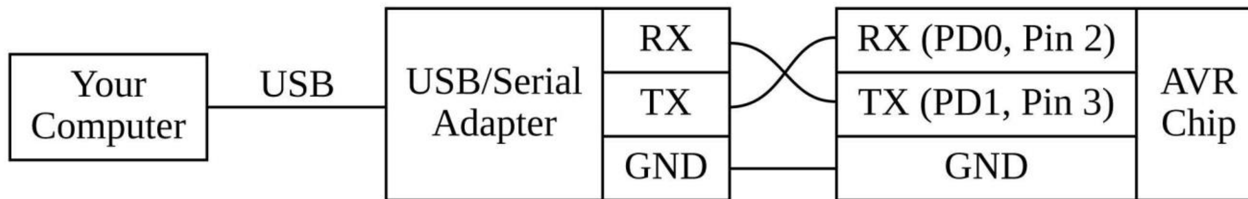
- Programming the Arduino since it is equipped with a bootloader.
- Serial communication with the Arduino IDE.
- Serial communication with other serial-enabled devices.

USART Serial

Transmit (TX) and Receive (RX) pins are connected **indirectly** to the transmit and receive lines of your USB cable.

Serial and USB are not directly compatible, and an intermediary USB-to-serial convertor needs to be used. This convertor can be an FTDI chip or a secondary USB-capable microcontroller like ATmega16U2 on the Arduino Uno.

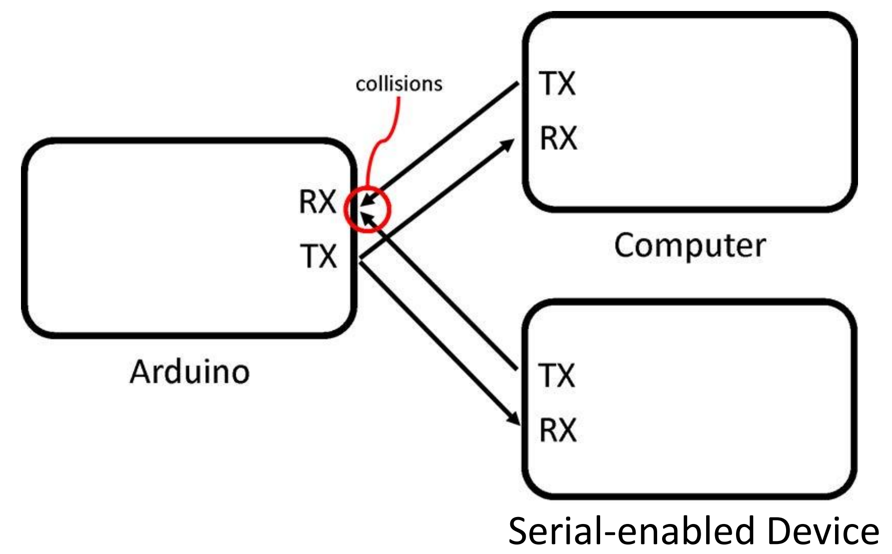




USART Serial

Since ATMega328P microcontroller (on the Arduino Uno) has only one hardware serial port (USART), you cannot program the Arduino or talk to it from your computer while another serial-enabled device is connected to the Arduino's serial port (RX and TX) at the same time.

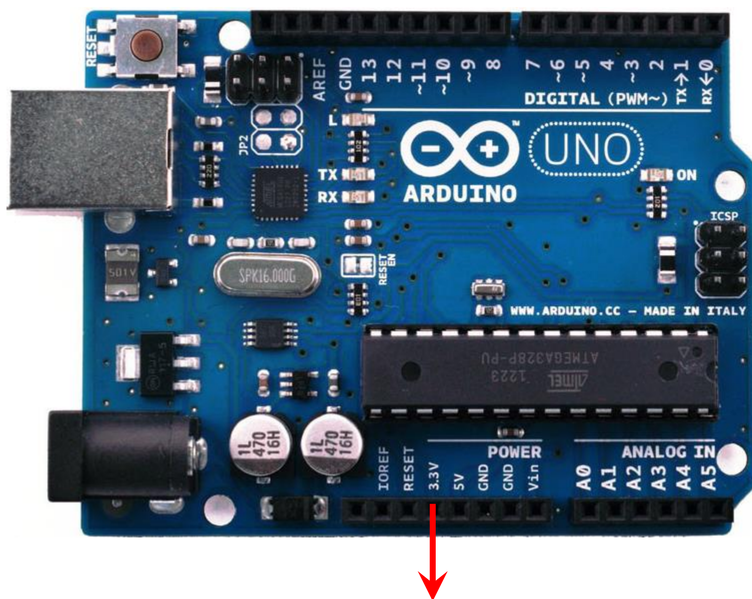
A **solution** is to use **SoftwareSerial.h** library that allows you to define two arbitrary digital pins on your Arduino to act as RX/TX pins for talking with another serial-enabled device (using software to replicate the functionality of the hardwired RX and TX lines).



Note that if you use serial communication with computer, you cannot also use pins 0 and 1 for digital input or output.

Level Conversion

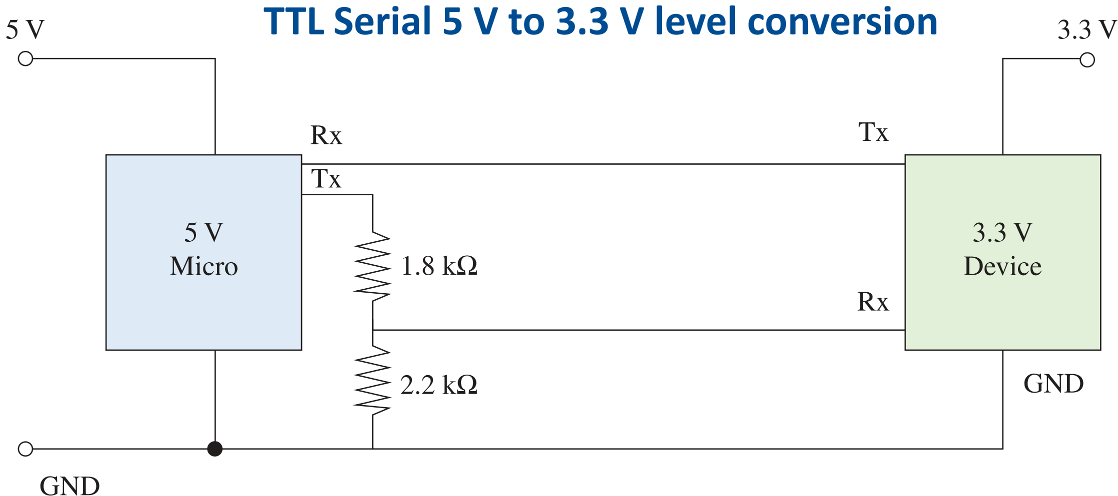
There is a recent trend for microcontrollers and other digital ICs to use 3.3 V or even 1.8 V rather than 5 V. Lower-voltage devices use less current and can be more convenient to power from batteries. When you are communicating with the ICs using one of the bus and serial interfaces, you will need to make sure that you convert voltage levels appropriately.



3.3V Output

SPI & USART Serial Level Conversion

Converting levels on SPI and TTL Serial is quite easy, because they have separate lines for each direction of communication. You only need to use a simple voltage divider.

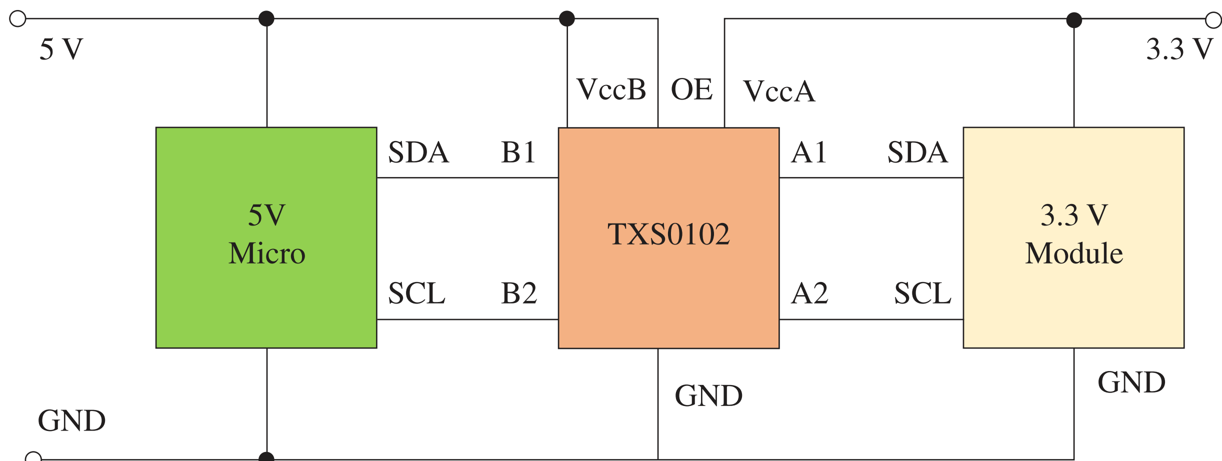


The **Tx** output of the 3.3 V device can be connected directly to the **Rx** input of the 5 V microprocessor, because it will see any input over about 3 V as a logical high anyway. The voltage divider is required when the 5 V **Tx** output of the microprocessor must be reduced to prevent damage to the 3.3 V device.

I^2C & 1-Wire Level Conversion

The problem is more complex when pins change modes, from being an input and being an output, as they do with I^2C and 1-Wire. In both these cases, the best solution is to use a custom level-shifting IC, which can convert two levels.
Recommended ICs: TXS0102, MAX3372, PCA9509, and PCA9306.

TXS0102 level converter used for I^2C

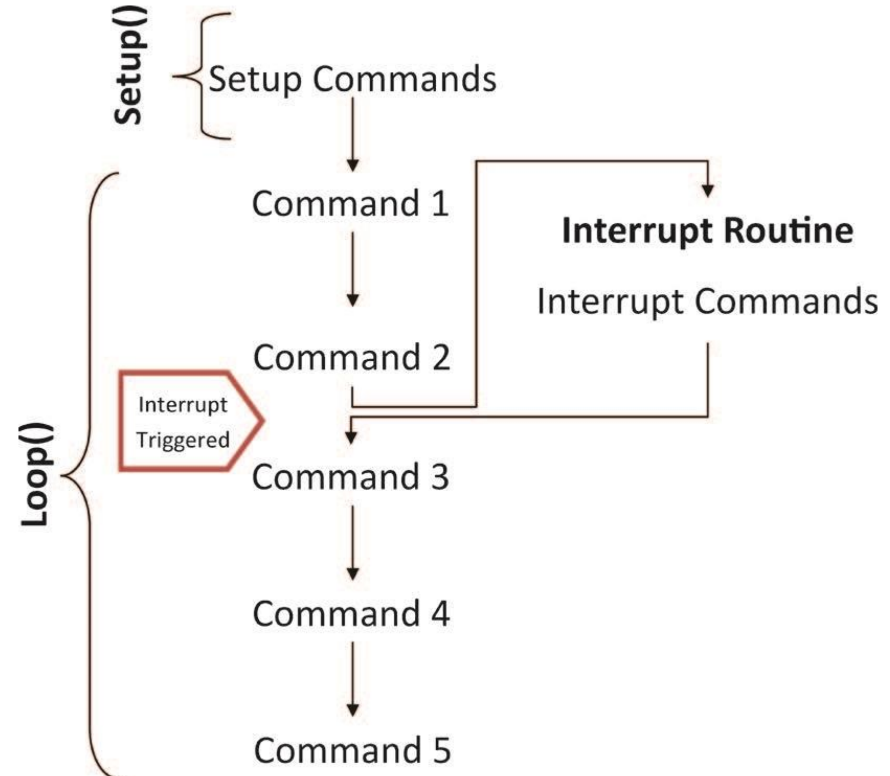


(OE: output-enable)

Interrupts

Interrupts allow you to stop whatever your Arduino is currently doing, complete a different task, and then return to what the Arduino was previously executing.

- **Hardware** (or External) **Interrupts**
- **Timer** (or Internal) **Interrupts**



Hardware Interrupts

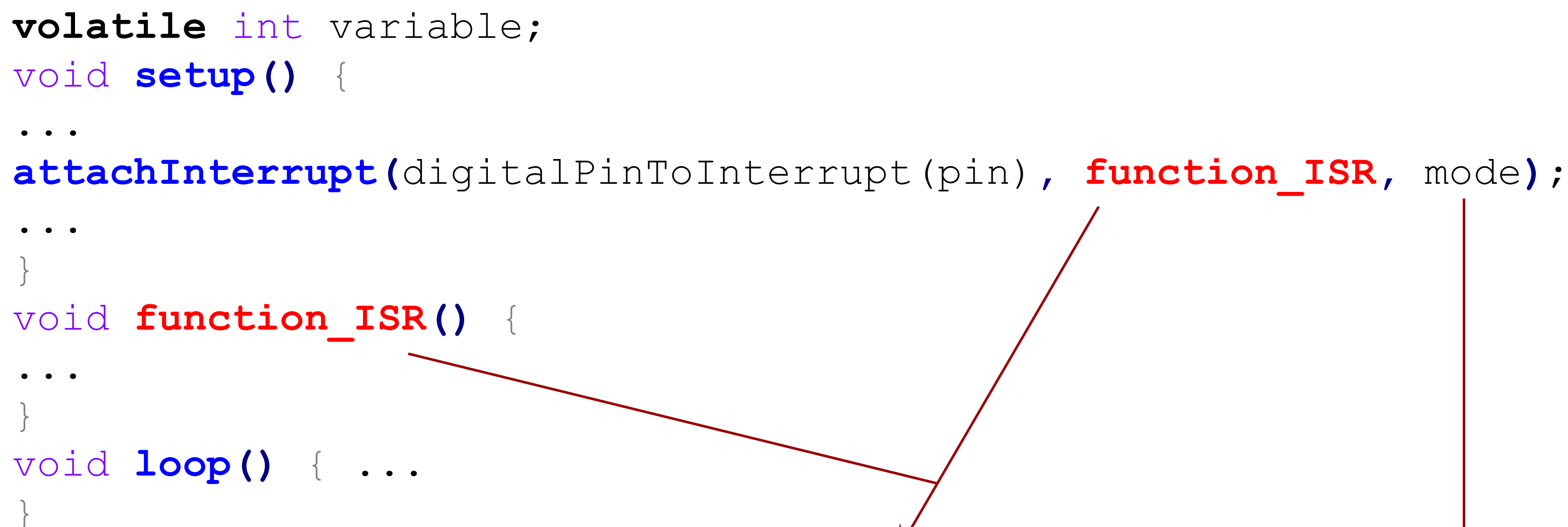
Hardware Interrupts are trigged depending on the state (or change in state), of an input I/O pin. Hardware interrupts can be particularly useful if you want to change some state variable within your code without having to constantly poll the state of a button. Hence, you can execute your main program, and have it “interrupted” to run a special function whenever an external interrupt event is detected. This interrupt can happen anywhere in the program’s execution.

Pin 2: INT 0
Pin 3: INT 1

ATMega328P microcontroller (on the Arduino Uno) supports just two external interrupts (INT 0 and INT 1).

❖ For certain fast acquisition tasks like using a rotary encoder, interrupting is an absolute necessity.

Hardware Interrupts



- ISR (Interrupt Service Routine) is a function which is called when the interrupt occurs.
- ISR must take no parameters and return nothing.
- ISR should be as short and fast as possible.

To make sure global variables shared between an ISR and the main program are updated correctly, declare them as **volatile**.

mode defines when the interrupt should be triggered: LOW, CHANGE, RISING, FALLING, HIGH

Timer Interrupts

The ATMega328P (on the Arduino Uno) has three hardware timers, which can be used to increment millis(), operate delay(), and enable PWM output with analogWrite(). You can also take manual control of one of these timers to initiate timed functions (triggering a function every set number of microseconds), generate arbitrary PWM signals on any pin, and more.

A third-party library (the **TimerOne.h** library) is used to take manual control of the 16-bit Timer1 on the ATMega328-based Arduinos.